SANDIA REPORT

SAND2014-4968
Unlimited Release
Printed June, 2014

UQTk Version 2.1 User Manual

Bert Debusschere, Khachik Sargsyan, Cosmin Safta

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2014-4968
Unlimited Release
Printed June, 2014

UQTk Version 2.1 User Manual

Bert Debusschere, Khachik Sargsyan, Cosmin Safta

Abstract

The UQ Toolkit (UQTk) is a collection of libraries and tools for the quantification of uncertainty in numerical model predictions. Version 2.1 offers intrusive and non-intrusive methods for propagating input uncertainties through computational models, tools for sensitivity analysis, methods for sparse surrogate construction, and Bayesian inference tools for inferring parameters from experimental data. This manual discusses the download and installation process for UQTk, provides pointers to the UQ methods used in the toolkit, and describes some of the examples provided with the toolkit.

Contents

Chapter 1

Overview

The UQ Toolkit (UQTk) is a collection of libraries and tools for the quantification of uncertainty in numerical model predictions. In general, uncertainty quantification (UQ) pertains to all aspects that affect the predictive fidelity of a numerical simulation, from the uncertainty in the experimental data that was used to inform the parameters of a chosen model, and the propagation of uncertain parameters and boundary conditions through that model, to the choice of the model itself.

In particular, UQTk provides implementations of many probabilistic approaches for UQ in this general context. Version 2.1 offers intrusive and non-intrusive methods for propagating input uncertainties through computational models, tools for sensitivity analysis, methods for sparse surrogate construction, and Bayesian inference tools for inferring parameters from experimental data.

The main objective of UQTk is to make these methods available to the broader scientific community for the purposes of algorithmic development in UQ or educational use. The most direct way to use the libraries is to link to them directly from C++ programs. Alternatively, in the examples section, many scripts for common UQ operations are provided, which can be modified to fit the users' purposes using existing numerical simulation codes as a black-box.

The next chapter in this manual discusses the download and installation process for UQTk, followed by some pointers to the UQ methods used in the toolkit, and a description of some of the examples provided with the toolkit.

Chapter 2

Download and Installation

Requirements

The core UQTk libraries are written in C++, with some dependencies on FORTRAN numerical libraries. As such, to use UQTk, a compatible C++ and FORTRAN compiler will be needed. UQTk is installed and built most naturally on a Unix-like platform, and has been tested on Mac OS X and Linux. Installation and use on Windows machines under Cygwin is possible, but has not been tested.

Many of the examples rely on Python and matplotlib for postprocessing and graphing. As such, Python version 2.7.x with compatible NumPy, SciPy, and matplotlib are recommended. Further the use of XML for input files requires the Expat XML parser library to be installed on your system. Note, if you will be linking the core UQTk libraries directly to your own codes, and do not plan on using the UQTk examples, then those additional dependencies are not required.

Download

The most recent version of UQTk, currently 2.1, can be downloaded from the following location:
`http://www.sandia.gov/UQToolkit`

After download, extract the tar file into the directory where you want to install UQTk.

```
% tar -xzvf uqtk_src_v2.0_Oct-14-2013.tgz
```

Make sure to replace the name of the tar file in this command with the name of the most recent tar file you just downloaded.

Directory Structure

After extraction, there will be a new directory `UQTk_v2.0` (version number may be different). Inside this top level directory are the following directories:

`config`	Configuration files
`doc_cpp`	Documentation for C++ libraries
`src_cpp`	C++ source code
`examples_cpp`	Examples with C++ libraries
`pyUQTk`	Python scripts
`src_matlab`	Matlab toolbox
`examples_matlab`	Matlab examples

Compilation

Before compiling, some configuration settings need to specified, such as the location of your compilers. To do so, change your directory to the configuration directory (again change the version number in the directory name below to the current version):

```
% cd UQTk_v2.1/config
```

In this directory, create a file named `config.site`, based on one of the templates provided. *E.g.*, if you are working on a Mac or Linux machine with the GNU compilers installed, the file `config.gnu` may be a good place to start from. Copy the file over to config.site, and edit the paths to point to the compiler locations on your system.

Additionally, to use some of the scripts in the examples provided with the distribution, the path to the upper level directory of UQTk needs to be set in the environment variable `UQTK_SRC`. If you are using the tcsh shell, and you extracted the tar file into the directory `~/software` *e.g.*, then set the environment variable as follows:

```
% setenv UQTK_SRC ~/software/UQTk_v2.1
```

or the bash shell:

```
% UQTK_SRC=~/software/UQTk_v2.1
% export UQTK_SRC
```

Test the value with `echo $UQTK_SRC`.

Further, to use some of Python scripts, the location of the pyUQTk directory needs to be in the Python search path, which is stored in the `PYTHONPATH` environment variable. If you are using the tcsh shell:

```
% setenv PYTHONPATH "~/software/UQTk_v2.1:${PYTHONPATH}"
```

or with the bash shell:

```
% PYTHONPATH="~/software/UQTk_v2.1:${PYTHONPATH}"
% export PYTHONPATH
```

When this is done, go back up to the main level directory and compile with the `make` command.

```
% cd ..
% make
```

If all goes well, there should be no errors. After compilation ends, there will be three new directories in the `src_cpp` directory:

`src_cpp/lib`	Compiled library files
`src_cpp/include`	Include files for all libraries
`src_cpp/bin`	Binary executables for the apps that come with UQTk

To use the UQTk libraries, your program should link in the libraries in `src_cpp/lib` and add the `src_cpp/include` directory to the compiler include path. The apps are standalone programs that perform UQ operations, such as response surface construction, or sampling from random variables. For more details, see the Examples section.

Chapter 3

Source Code Description

UQTk implements many probabilistic methods found in the literature. For more details on the methods, please refer to the following papers and books on Polynomial Chaos methods for uncertainty propagation [2, 4], Bayesian inference [5], Bayesian compressive sensing [1], and the Rosenblatt transformation [6].

For more details on the actual source code in UQTk, HTML documentation is also available in the `doc_cpp/html` folder.

Chapter 4

Examples

The primary intended use for UQTk is as a library that provides UQ functionality to numerical simulations. To aid the development of UQ-enabled simulation codes, some examples of programs that perform common UQ operations with UQTk are provided with the distribution. These examples can serve as a template to be modified for the user's purposes. In some cases, *e.g.* in sampling-based approaches where the simulation code is used as a black-box entity, the examples may provide enough functionality to be used directly, with only minor adjustments. Below is a brief description of the main examples that are currently in the UQTk distribution. For all of these, make sure the environment variable `UQTK_SRC` is set and points to the UQTk upper level directory (*i.e.* the one that has `src_cpp` and `examples_cpp` as subdirectories), as described in the compilation section.

Elementary Operations

- Located in `examples_cpp/ops`
- Illustrates the use of UQTk for elementary operations on random variables that are represented with Polynomial Chaos (PC) expansions.
- To run an example, type `make examples` in `examples_cpp/ops` or run `./prob1.py`
- For more documentation, see `examples_cpp/ops/prob1.pdf`

Forward Propagation of Uncertainty

- Located in `examples_cpp/surf_rxn`
- Several examples of propagating uncertainty in input parameters through a model for surface reactions, consisting of three Ordinary Differential Equations (ODEs). Two approaches are illustrated:
  - Direct linking to the C++ UQTk libraries from a C++ simulation code:

    * Propagation of input uncertainties with Intrusive Spectral Projection (ISP), Non Intrusive Spectral Projection (NISP) via quadrature , and NISP via Monte Carlo (MC) sampling.
    * For more documentation, see `examples_cpp/surf_rxn/prob2.pdf`
    * An example can be run with `./prob2.py`
  - Using simulation code as a black box forward model:
    * Propagation of uncertainty in one input parameter with NISP quadrature approach.
    * For more documentation, see `examples_cpp/surf_rxn/prob3.pdf`
    * An example can be run with `./prob3.py`

- Both examples can be run with `make examples` in `examples_cpp/surf_rxn`

Bayesian Inference of a Line

Overview

This example is located in `examples_cpp/line_infer` It infers the slope and intercept of a line from noisy data using Bayes' rule. The C++ libraries are called directly from the driver program. By changing the likelihood function and the input data, this program can be tailored to other inference problems.

To run an example, type `make examples` in `examples_cpp/line_infer` or run `./line_infer.py` directly.

The file `line_infer.py` contains quite a bit of inline documentation about the various settings and methods used. To get a listing of all command line options, type `./line_infer.py -h"` A typical run is as follows:

```
./line_infer.py --nd 9 --stats
```

This will run the inference problem with 9 data points, generate plots of the posterior distributions, and generate statistics of the MCMC samples. If no plots are desired, also give the `--noplots` argument.

More details

After setting a number of default values for the example problem overall, the `line_infer.py` script sets the proper inference inputs in the file `line_infer.xml`, starting from a set of defaults in `line_infer.xml.templ`. The file `line_infer.xml` is read in by the C++ code `line_infer.x`, which does the actual Bayesian inference. After that, synthetic data is generated, either from a linear, or cosine model, with added noise.

Then, the script calls the C++ line inference code `line_infer.x` to infer the two parameters (slope and intercept) of a line that best fits the artificial data. (Note, one can also run the inference code directly by manually editing the file `line_infer.xml` and typing the command `./line_infer.x`)

The script then reads in the MCMC posterior samples file, and performs some postprocessing:

Unless the flag `--noplots` is specified, the script computes and plots the following:

- The pushed-forward and posterior predictive error bars
  - Generate a dense grid of x-values
  - Evaluate the linear model $y = a + bx$ for all posterior samples (a, b) after the burn-in
  - Pushed-forward distribution: compute the sample mean and standard deviation of using the sampled models
  - Posterior predictive distribution: combine pushed-forward distribution with the noise model
- The MCMC chain for each variable, as well as a scatter plot for each pair of variables
- The marginal posterior distribution for each variable, as well as the marginal joint distribution for each pair of variables

If the flag `--stats` is specified, the following statistics are also computed:

- The mean, MAP (Maximum A Posteriori), and standard deviations of all parameters
- The covariance matrix
- The average acceptance probability of the chain
- The effective sample sizes for each variable in the chain

Sample Results

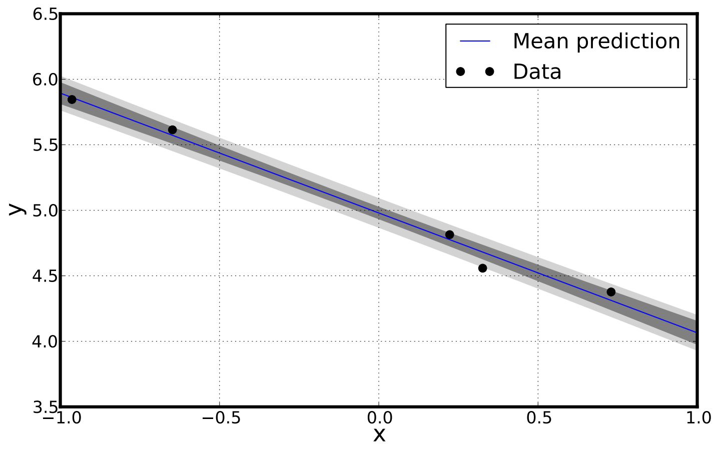

Figure 4.1. The pushed forward posterior distribution (dark grey) and posterior predictive distribution (light grey).

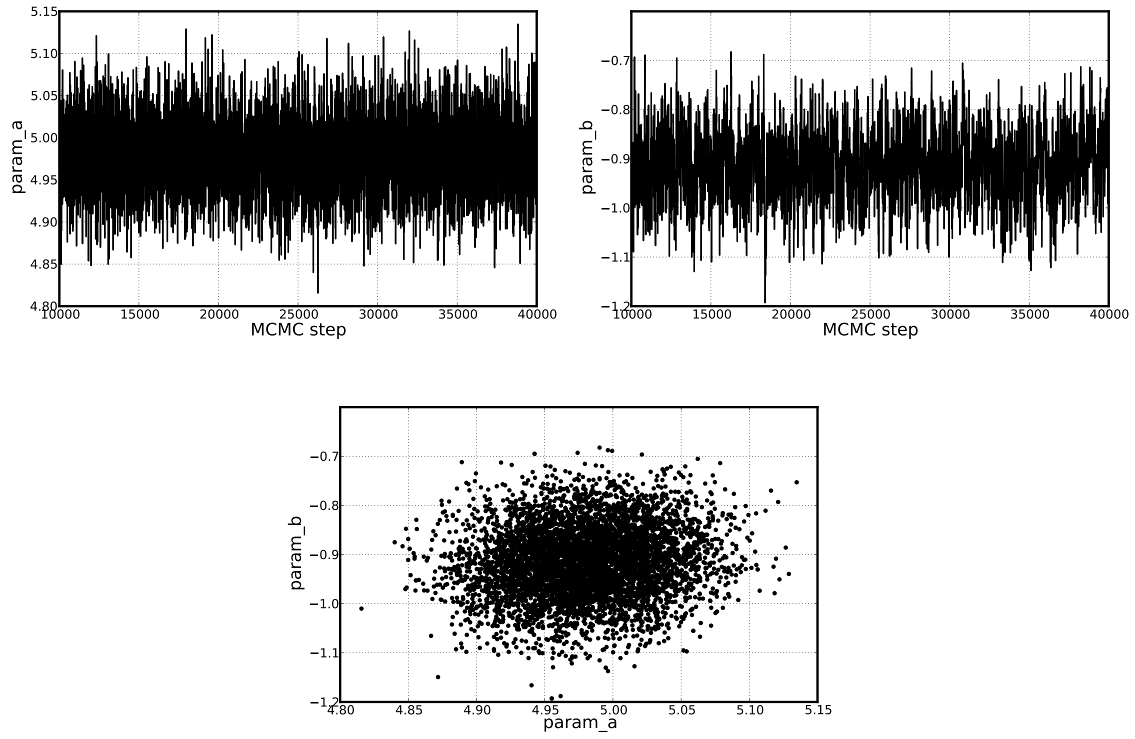

Figure 4.2. MCMC chains for parameters a and b, as well as a scatter plot for a and b

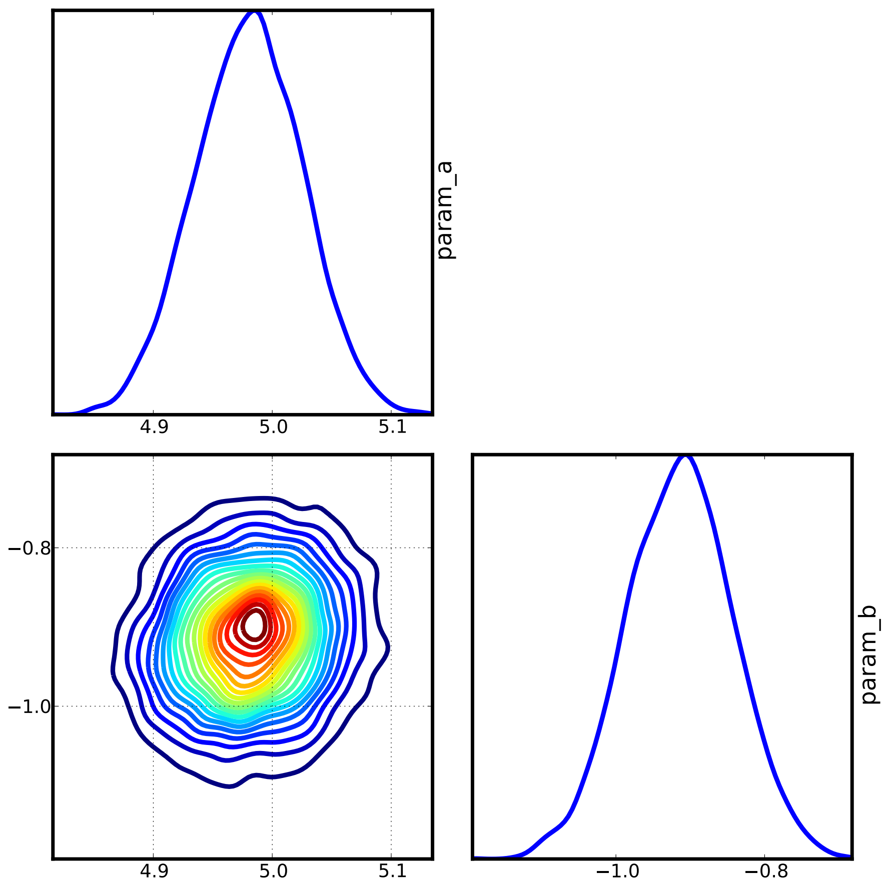

Figure 4.3. Marginal posterior distributions for all variables, as well as marginal joint posteriors

Surrogate Construction and Sensitivity Analysis

Overview

- Located in `examples_cpp/uq_surr`
- A collection of scripts that construct a PC surrogate for a computational model which is specified as a black box simulation code. Also provides tools for sensitivity analysis of the outputs of this black box model with respect to its input parameters.

Theory

Consider a function $f(\lambda; x)$ where $\lambda = (\lambda_1, \dots, \lambda_d)$ are the model *input* parameters of interest, while $x \in \mathbb{R}^r$ are *design* parameters with controllable values. For example, x can denote a spatial coordinate or a time snapshot, or it can simply enumerate multiple quantities of interest. Furthermore, we assume known domains for each input parameter, i.e.

$$\lambda_i \in [a_i, b_i] \qquad \text{for} \quad i = 1, \dots .d. \tag{4.1}$$

The goal is to build a Polynomial Chaos (PC) *surrogate* function for each value of design parameter x, i.e. for $l = 1, \dots, L$,

$$f(\lambda; x_l) \approx g(\lambda; x_l) = \sum_{k=0}^{K-1} c_{kl} \Psi_k(\xi) \tag{4.2}$$

with respect to scaled inputs

$$\xi_i = \frac{\lambda_i - \frac{b_i - a_i}{2}}{\frac{b_i + a_i}{2}} \in [-1, 1] \qquad \text{for} \quad i = 1, \dots .d. \tag{4.3}$$

Here $\Psi_k(\xi) = \Psi_k(\xi_1, \dots, \xi_d)$ are multivariate normalized Legendre polynomials, defined as products of univariate normalized Legendre polynomials $\psi_{k_i}(\xi_i)$ as follows:

$$\Psi_k(\xi) = \psi_{k_1}(\xi_1) \dots \psi_{k_d}(\xi_d). \tag{4.4}$$

A typical truncation rule in (4.2) is defined according to the total order of the basis terms, i.e. only polynomials with the total order $\leq p$ are retained for some positive integer order p, implying $|k_1| + \cdots + |k_d| \leq p$, and $K = (d+p)!/(d!p!)$. The scalar index k is simply counting the *multi-indices* $(k_1, \dots, k_d)$.

After computing the PC coefficients c_{kl}, one can extract the global sensitivity information, also called Sobol indices or variance-based decomposition. For example, the *main sensitivity* index with respect to the dimension i (or variable ξ_i) is

$$S_i(x_l) = \frac{\sum_{k \in \mathbb{I}_i} c_{kl}^2}{\sum_{k=1}^{K-1} c_{kl}^2}, \tag{4.5}$$

where $\mathbb{I}_i$ is the indices of basis terms that involve only the variable ξ_i, i.e. the one-dimensional monomials $\psi_1(\xi_i), \psi_2(\xi_i), ...$ In other words, these are basis terms corresponding to multi-indices with the only non-zero entry at the i-th location.

The two generic methods of finding the PC coefficients c_{kl} are detailed below.

Projection

The basis orthonormality enables the projection formulae

$$c_{kl} = \int_\Omega f(\lambda(\xi); x_l)\Psi_k(\xi)\frac{1}{2^d}d\xi \tag{4.6}$$

where $\Omega = [-1, 1]^d$ and $\lambda(\xi)$ simply denotes the linear scaling relation in (4.3).

The projection integral can be taken by

- Quadrature integration

$$c_{kl} \approx \sum_{q=1}^{Q} f(\lambda(\xi^{(q)}); x_l)\Psi_k(\xi^{(q)}), \tag{4.7}$$

  where $\xi^{(q)}$ are Gauss-Legendre quadrature points in the d-dimensional $\Omega = [-1, 1]^d$.

- Monte-Carlo integration [to be implemented]

$$c_{kl} \approx \sum_{m=1}^{M} f(\lambda(\xi^{(m)}); x_l)\Psi_k(\xi^{(m)}), \tag{4.8}$$

  where $\xi^{(m)}$ are uniformly random points in the d-dimensional $\Omega = [-1, 1]^d$.

Inference

[to be implemented]

Implementation

The script set consists of three files are

- `uq_pc.py` : the main script
- `model.py` : black-box example model

- `plot.py` : plotting

The apps employed

- `generate_quad` : Quadrature point/weight generation
- `gen_mi` : PC multiindex generation
- `pce_resp` : Projection via quadrature integration
- `pce_sens` : Sensitivity extraction
- `pce_eval` : PC evaluation

Main script: The syntax of the main script is

```
% uq_pc.py -r <run_regime>
%     -p <pdomain_file> -m <method> -o <ord> -s <sam_method> -n <nqd>  -v <nval>
```

Also one can run `uq_pc.py -h` for help in the terminal and to check the defaults.

The list of arguments:

- `-r <run_regime>`: The regime in which the workflow is employed. The options are

  `online` : Black-box model, which is in `model.py`, is run directly as parameter ensemble becomes available. User can provide their own `model.py` or simply use the current one as an example.

  `offline_prep` : Prepare the input parameter ensemble and store in `ytrain.dat` and, if validation is requested, `yval.dat`.
  The user then should run the model (`model.py ptrain.dat ytrain.dat` and perhaps `model.py pval.dat yval.dat`) in order to provide ensemble output for the `offline_post` stage.

  `offline_post` : Postprocess the output ensemble, assuming the model is run offline with input ensemble provided in the `offline_prep` stage producing `ytrain.dat` and, if validation is requested, `yval.dat`. The rest of the arguments should remain the same as in the `offline_post` stage.

- `-p <domain_file>`: A file with d rows and 2 columns, where d is the number of parameters and each row consists of the lower and upper bound of the corresponding parameter.

- `-m <method>`: The method of finding the PC surrogate coefficients. The options are

  `proj` : Projection method outlined in (4.6) and (4.7)

  `lsq` : Least-squares. To be implemented.

  `bcs` : Bayesian compressive sensing. To be implemented.

- `-o <ord>`: Total order p of the requested PC surrogates.
- `-s <sam_method>`: The input parameter sampling method. The options are

  `Q` : Quadrature points. This sampling scheme works with the projection method only, described in (4.7)

  `U` : Uniformly random points. To be implemented.
- `-n <nqd>`: Number of samples requested if `sam_method=U`, or the number of quadrature points per dimension, if `sam_method=Q`.
- `-v <nval>`: Number of uniformly random samples generated for PC surrogate validation, can be equal to 0 to skip validation.

Generated output files are:

- `allsens.dat` and `allsens_sc.dat` :The main effect sensitivity indices in a format Lxd, where each row corresponds to a single value for the design parameter, and each column corresponds to the sensitivity index of a parameter. The second file stores the same results, only the sensitivity indices are scaled to sum to one for each parameter.
- `results.pk` : Python pickle file that includes surrogate, sensitivity and error information. It is loaded in plotting utilities.
- `*.log` : Log files of the apps that have been employed, for reference.
- `mi.dat` : PC multiindex, for reference, in a matrix form of size Kxd, see (4.4).
- `designPar.dat` : A file containing values for the design parameters, for reference.

Black-box model: The syntax of the model script is

```
% model.py <modelPar_file>  <output_file>
```

Also one can run `model.py -h` for help in the terminal and to check the defaults.

The list of arguments:

- `<modelPar_file>` : A file with N rows and d columns that stores the input parameter ensemble of N samples.
- `<output_file>` : The file where output is stored, with N rows (number of input parameter samples) and L columns (number of outputs, or number of design parameter values).

`model.py` is the file that needs to be modified/provided by the user according to the model under study. Currently, a simple example function $f(\lambda; x) = \left(\sum_{i=1}^{d} \lambda_i\right)\left(\sum_{i=1}^{d} \frac{\lambda_i+\lambda_i^2}{i^{x+1}}\right)$ is implemented that also produces the file `designPar.dat` for design parameters $x_l = l$ for $l = 0, \ldots, 6$. The default dimensionality is set to $d = 3$.
A user-created black-box `model.py` should accept an input parameter file `<modelPar_file>` and should produce an output file `<output_file>` with the formats described above, in order to be consistent with the expected I/O.

Visualization: The syntax of the plotting script is

```
% plot.py <plotid>
```

The user is encouraged to enhance or change the visualization scripts. The current defaults, explained below, are simply for illustration purposes. The options for the only argument `<plotid>` are

`sens` : Plots the sensitivity information.

`dm` : Plots model-vs-data for one of the values of the design parameter.

`idm` : Plots model and data values on the same axis, for one of the values of the design parameter.

For the visualization script above, make sure the `PYTHONPATH` environment variable contains the directory `UQTK_SRC`.

Sample run:

In order to run a simple example, use the prepared an input domain file which is the default domain file, `pdomain_3d.dat`, and run

- Online mode: `uq_pc.py -r online -v111`

- Offline mode: `uq_pc.py -r offline_prep -v111`,
  followed by model evaluations
  `model.py ptrain.dat ytrain.dat` and `model.py pval.dat yval.dat`,
  and the postprocessing stage `uq_pc.py -r offline_post -v111`

After finishing, run `plot.py sens` or `plot.py dm` or `plot.py idm` to visualize some results.

Global Sensitivity Analysis via Sampling

Overview

- Located in `pyUQTk/sensitivity`
- A collection of python functions that generate input samples for black-box models, followed by functions that post-process model outputs to generate total, first-order, and joint effect Sobol indices

Theory

Let $X = (X_1, \cdots, X_n) : \Omega \to \mathcal{X} \subset \mathrm{I\!R}^n$ be an $n-$dimensional Random Variable in $L^2(\Omega, \mathcal{S}, P)$ with probability density $X \sim p_X(x)$. Let $x = (x_1, \cdots, x_n) \in \mathcal{X}$ be a sample drawn from this density, with $\mathcal{X} = \mathcal{X}_1 \otimes \mathcal{X}_2 \otimes \cdots \otimes \mathcal{X}_n$, and $\mathcal{X}_i \subset \mathrm{I\!R}$ is the range of X_i.

Let $X_{-i} = (X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_n) : \Omega \to \mathcal{X}_{-i} \subset \mathrm{I\!R}^{n-1}$, where $X_{-i} \sim p_{X_{-i}|X_i}(x_{-i}|x_i) = p_X(x)/p_{X_i}(x_i)$, $p_{X_i}(x_i)$ is the marginal density of X_i, $x_{-i} = (x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$, and $\mathcal{X}_{-i} = \mathcal{X}_1 \otimes \cdots \otimes \mathcal{X}_{i-1} \otimes \mathcal{X}_{i+1} \otimes \cdots \otimes \mathcal{X}_n$.

Consider a function $Y = f(X) : \Omega \to \mathrm{I\!R}$, with $Y \in L^2(\Omega, \mathcal{S}, P)$. Further, let $Y \sim p_Y(y)$, with $y = f(x)$. Given the variance of f is finite , one can employ the law of total variance[1,2] to decompose the variance of f as

$$V[f] = V_{x_i}[E[f|x_i]] + E_{x_i}[V[f|x_i]] \tag{4.9}$$

The conditional mean, $E[f|x_i] \equiv E[f(X)|X_i = x_i]$, and conditional variance, $V[f|x_i] = V[f(X)|X_i = x_i]$, are defined as

$$\begin{aligned}
\langle f \rangle_{-i} \equiv E[f|x_i] &= \int_{\mathcal{X}_{-i}} f(x) p_{X_{-i}|X_i}(x_{-i}|x_i) dx_{-i} &(4.10)\\
V[f|x_i] &= E[(f - \langle f \rangle_{-i})^2|x_i] \\
&= E[(f^2 - 2f\langle f \rangle_{-i} + \langle f \rangle_{-i}^2)|x_i] \\
&= E[f^2|x_i] - 2\langle f \rangle_{-i}\langle f \rangle_{-i} + \langle f \rangle_{-i}^2 \\
&= \int_{\mathcal{X}_{-i}} f(x)^2 p_{X_{-i}|X_i}(x_{-i}|x_i) dx_{-i} - \langle f \rangle_{-i}^2 &(4.11)
\end{aligned}$$

[1] `en.wikipedia.org/wiki/Law_of_total_variance`

[2] `en.wikipedia.org/wiki/Law_of_total_expectation`

The terms in the *rhs* of Eq. (4.9) can be written as

$$\begin{aligned} V_{x_i}[E[f|x_i]] &= E_{x_i}[\ (E[f|x_i] - E_{x_i}[E[f|x_i]])^2\ ] \\ &= E_{x_i}[\ (E[f|x_i] - f_0)^2\ ] \\ &= E_{x_i}[\ (E[f|x_i])^2\ ] - f_0^2 \\ &= \int_{\mathcal{X}_i} E[f|x_i]^2 p_{X_i}(x_i) dx_i - f_0^2 \end{aligned} \tag{4.12}$$

where $f_0 = E[f] = E_{x_i}[E[f|x_i]]$ is the expectation of f, and

$$E_{x_i}[V[f|x_i]] = \int_{\mathcal{X}_i} V[f|x_i] p_{X_i}(x_i) dx_i \tag{4.13}$$

The ratio

$$S_i = \frac{V_{x_i}[E[f|x_i]]}{V[f]} \tag{4.14}$$

is called the first-order Sobol index [8] and

$$S_{-i}^T = \frac{E_{x_i}[V[f|x_i]]}{V[f]} \tag{4.15}$$

is the total effect Sobol index for x_{-i}. Using Eq. (4.9), the sum of the two indices defined above is

$$S_i + S_{-i}^T = S_{-i} + S_i^T = 1 \tag{4.16}$$

Joint Sobol indices S_{ij} are defined as

$$S_{ij} = \frac{V_{x_i,x_j}[E[f|x_i,x_j]]}{V[f]} - S_i - S_j \tag{4.17}$$

for $i, j = 1, 2 \ldots, n$ and $i \neq j$.

S_i can be interpreted as the fraction of the variance in model f that can be attributed to the i-th input parameter only, while S_{ij} is the variance fraction that is due to the joint contribution of i-th and j-th input parameters. S_i^T measures the fractional contribution to the total variance due to parameter x_i and its interactions with all other model parameters.

The Sobol indices are numerically estimated using Monte Carlo (MC) algorithms proposed by Saltelli [7] and Kucherenko *et al* [3]. Let $x^k = (x_1, \cdots, x_n)^k$ be a sample of X drawn from p_X. Let x'^k_{-i} be a sample from the conditional distribution $p_{X_{-i}|X_i}(x'_{-i}|x_i^k)$, and x''^k_i a sample from the conditional distribution $p_{X_i|X_{-i}}(x''_i|x^k_{-i})$.

The expectation $f_0 = E[f]$ and variance $V = V[f]$ are estimated using the x^k samples as

$$f_0 \approx \frac{1}{N}\sum_{k=1}^{N} f(x^k), \quad V \approx \frac{1}{N}\sum_{k=1}^{N} f(x^k)^2 - f_0^2 \tag{4.18}$$

where N is the total number of samples. The first-order Sobol indices S_i are estimated as

$$S_i \approx \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x'^k_{-i} \cup x^k_i) - f_0^2\right) \tag{4.19}$$

The joint Sobol indices are estimated as

$$S_{ij} \approx \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x'^k_{-(i,j)} \cup x^k_{i,j}) - f_0^2\right) - S_i - S_j \tag{4.20}$$

For S_i^T, UQTk offers two alternative MC estimators. In the first approach, S_i^T is estimated as

$$S_i^T = 1 - S_{-i} \approx 1 - \frac{1}{V}\left(\frac{1}{N}\sum_{k=1}^{N} f(x^k) f(x''^k_i \cup x^k_{-i}) - f_0^2\right) \tag{4.21}$$

In the second approach, S_i^T is estimated as

$$S_i^T \approx \frac{1}{2V}\left(\frac{1}{N}\sum_{k=1}^{N} \left(f(x^k) - f(x^k_{-i} \cup x''^k_i)\right)^2\right) \tag{4.22}$$

Implementation

Directory `pyUQTk/sensitivity` contains two python files

- `gsalib.py` : set of python functions implementing the MC sampling and estimators for Sobol indices
- `gsatest.py` : workflow illustrating the computation of Sobol indices for a toy problem

`gsalib.py` implements the following functions

- `genSpl_Si(nspl,ndim,abrng,**kwargs)` : generates samples for Eq. (4.19). The input parameters are as follows

  `nspl`: number of samples N,

  `ndim`: dimensionality n of the input parameter space ,

  `abrng`: a 2-dimensional array $n \times 2$, containing the range for each component x_i.

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples. These samples are used in subsequent calculations of joint Sobol indices

  `verb`: verbosity level

  `nd`: number of significant digits for ascii output

  The default values for optional parameters are listed in `gsalib.py`

- `genSens_Si(modeval,ndim,**kwargs)` : computes first-order Sobol indices using Eq. (4.19). The input parameters are as follows

  `modeval`: name of ascii file with model evaluations,

  `ndim`: dimensionality n of the input parameter space

  The following optional parameter can also be specified

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

- `genSpl_SiT(nspl,ndim,abrng,**kwargs)` : generates samples for Eqs. (4.21-4.22). The input parameters are as follows

  `nspl`: number of samples N,

  `ndim`: dimensionality n of the input parameter space ,

  `abrng`: an 2-dimensional array $n \times 2$, containing the range for each component x_i.

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples. These samples are used in subsequent calculations of Sobol indices

  `verb`: verbosity level

  `nd`: number of significant digits for ascii output

  The default values for optional parameters are listed in `gsalib.py`

- `genSens_SiT(modeval,ndim,**kwargs)` : computes total Sobol indices using either Eq. (4.21) or Eq. (4.22). The input parameters are as follows

  `modeval`: name of ascii file with model evaluations,

  `ndim`: dimensionality n of the input parameter space

  The following optional parameter can also be specified

  `type`: specifies wether to use Eq. (4.21) for type = "type1" or Eq. (4.22) for type $\neq$ "type1"

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

- `genSpl_Sij(ndim,**kwargs)` : generates samples for Eq. (4.20). The input parameters are as follows

  `ndim`: dimensionality n of the input parameter space ,

  The following optional parameters can also be specified

  `splout`: name of ascii output file for MC samples

  `matfile`: name of binary output file for select MC samples saved by `genSpl_Si`.

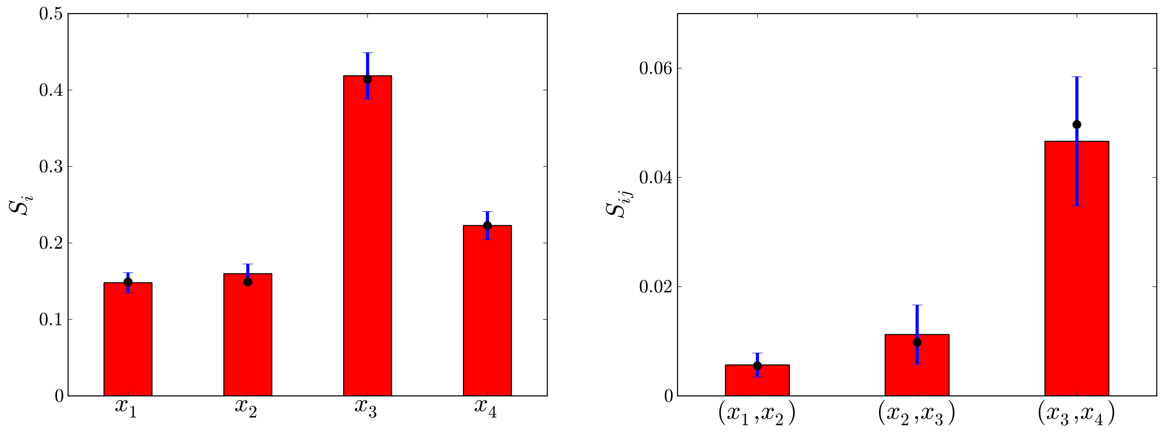

Figure 4.4. First-order (left frame) and joint (right frame) Sobol indices for the model given in Eq. (4.23). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on an ensemble of 10 runs.

`verb`: verbosity level

`nd`: number of significant digits for ascii output

The default values for optional parameters are listed in `gsalib.py`

- `genSens_Sij(sobolSi,modeval,**kwargs)` : computes joint Sobol indices using Eq. (4.20). The input parameters are as follows

  `sobolSi`: array with values for first-order Sobol indices S_i

  `modeval`: name of ascii file with model evaluations.

  The following optional parameter can also be specified

  `verb`: verbosity level

  The default value for the optional parameter is listed in `gsalib.py`

`gsatest.py` provides the workflow for the estimation of Sobol indices for a simple model given by

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n-1} i^2 x_i x_{i+1} \tag{4.23}$$

In the example provided in this file, n (`ndim` in the file) is set equal to 4, and the number of samples N (`nspl` in the file) to 10^4. Figures 4.4 and 4.5 show results based on an ensemble of 10 runs. To generate these results run the example workflow:

```
python gsatest.py
```

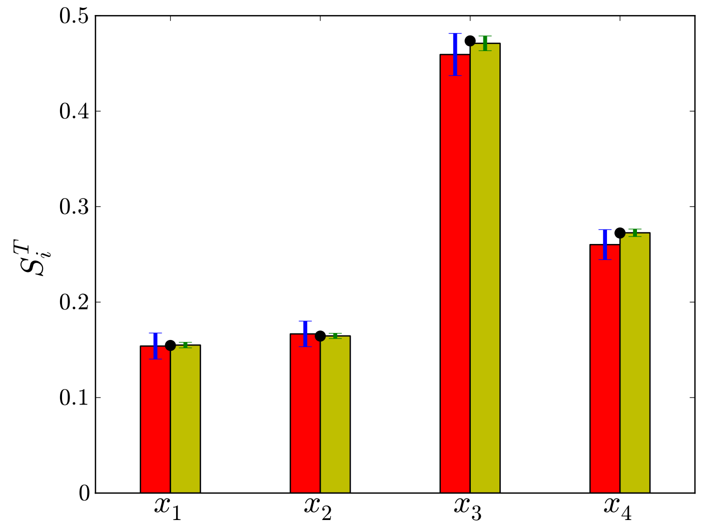

Figure 4.5. Total-order Sobol indices for the model given in Eq. (4.23). The red bars shows results based on Eq. (4.21) while the yellow bars are based on Eq. (4.22). The black circles show the theorerical values, computed analytically, and the error bars correspond to $\pm\sigma$ computed based on an ensemble of 10 runs. For this model, Eq. (4.22) provides more accurate estimates for S_i^T compared to results based on Eq. (4.21).

Karhunen-Loève Expansion of a Stochastic Process

- Located in `examples_cpp/kl_sample`
- Some examples of the construction of 1D and 2D Karhunen-Loève (KL) expansions of a Gaussian stochastic process, based on sample realizations of this stochastic process.
- For more information and examples, see `examples_cpp/kl_sample/kl_example.pdf`

Chapter 5

Support

UQTk is the subject of continual development and improvement. If you have questions about or suggestions for UQTk, feel free to e-mail Bert Debusschere, at `mailto:bjdebus@sandia.gov`.

References

[1] S. Babacan, R. Molina, and A. Katsaggelos. Bayesian compressive sensing using Laplace priors. *IEEE Transactions on Image Processing*, 19(1):53–63, 2010.

[2] B.J. Debusschere, H.N. Najm, P.P. Pébay, O.M. Knio, R.G. Ghanem, and O.P. Le Maître. Numerical challenges in the use of polynomial chaos representations for stochastic processes. *SIAM Journal on Scientific Computing*, 26(2):698–719, 2004.

[3] S. Kucherenko, S. Tarantola, and P. Annoni. Estimation of global sensitivity indices for models with dependent variables. *Computer Physics Communications*, 183:937–946, 2012.

[4] O.P. Le Maître and O.M. Knio. *Spectral Methods for Uncertainty Quantification: With Applications to Computational Fluid Dynamics (Scientific Computation)*. Springer, 1st edition. edition, April 2010.

[5] Y. M. Marzouk and H. N. Najm. Dimensionality reduction and polynomial chaos acceleration of Bayesian inference in inverse problems. *Journal of Computational Physics*, 228(6):1862–1902, 2009.

[6] M. Rosenblatt. Remarks on a multivariate transformation. *Annals of Mathematical Statistics*, 23(3):470 – 472, 1952.

[7] A. Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145:280–297, 2002.

[8] I. M. Sobol. Sensitivity estimates for nonlinear mathematical models. *Math. Modeling and Comput. Exper.*, 1:407–414, 1993.

DISTRIBUTION:

1	MS 0899	Technical Library, 8944 (electronic copy)